

COMMON LISP OBJECT SYSTEM

Exposition des objets

En utilisant le système d'objet, on a représenté plusieurs notations graphiques comme triangles, rectangle, carré, etc. Le principe est de créer un objet qui représente la racine d'objet, ensuite on crée d'autre classe par l'héritage et l'extension de cette classe.

Représenter ces objets graphiques en CLOS, en exploitant les conventions exposées ci-avant

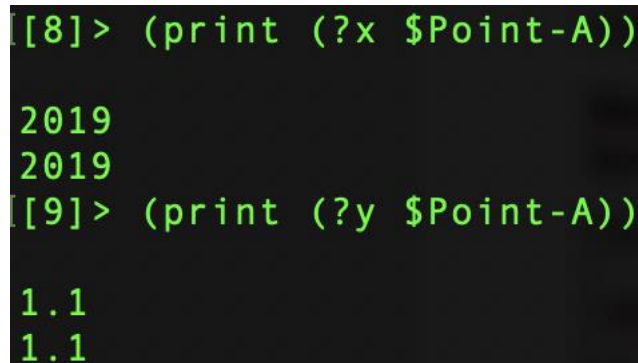
On départ avec un point qui a une propriété Nom et deux autres propriétés X et Y

```
(defclass $Point()  
  (  
    ($x :type real :accessor ?x :initarg :x) ($y :type real :accessor ?y :initarg :y).  
    ($Nom :accessor ?Nom :initarg :Nom)  
  )  
)
```

Maintenant on a créé une classe, on peut créer un objet de cette classe.

```
(setq $Point-A (make-instance '$Point :x 2019 :y 1.1))
```

Test sous terminal:



```
[[8]> (print (?x $Point-A))  
2019  
2019  
[[9]> (print (?y $Point-A))  
1.1  
1.1
```

Car un cercle vient d'un point avec un rayon, donc on peut hériter la classe du Point.

```
(defclass $Cercle ($Point)
  (
    ($r :accessor ?r :initarg :r)
  )
)
```

Avec la même idée, on peut créer les classes du Carre, Rectangle et Triangle.

```
defclass $Carre()
  (
    ($Sommet :type $Point :accessor ?Sommet :initarg :Sommet)
    ($w :accessor ?w :initarg :w)
  )
)
Ou
;; (defclass $Carre ($Point)
;;   (($w :accessor w? :initarg :w))
;; )

(defclass $Rectangle($Carre)
  (
    ($h :accessor ?h :initarg :h)
  )
)

(defclass $Triangle ()
  (
    ($M1 :type $Point :accessor ?M1 :initarg :M1)
    ($M2 :type $Point :accessor ?M2 :initarg :M2)
    ($M3 :type $Point :accessor ?M3 :initarg :M3)
  )
)
```

Pour la class Polygone est un peu différent, on sait bien que le Polygone peut présenter par une liste des points, donc on a :

```
(defclass $Polygone ()
  (
    ($sommets :type list :accessor ?sommets :initarg :sommets)
  )
)
```

Pour chaque objet graphique, définir les messages

On crée des méthodes pour chaque classe afin d'appliquer des transformations.

(a) **Translate (dx dy)** : demande À l'objet de se traduire du vecteur de coordonnées dx et dy.

Idée est simple : modifier ses coordonnées

Exemple de translate d'un point est Triangle :

```
(defmethod =translate ((p $Point) dx dy)
  (setf (?x p) (+ dx (?x p)))
  (setf (?y p) (+ dy (?y p)))
  (describe p)
)

(defmethod =translate((t $Triangle) dx dy)
  (=translate (?M1 t) dx dy)
  (=translate (?M2 t) dx dy)
  (=translate (?M3 t) dx dy)
)
```

Pour translate un Polygone on doit traduire tous les point :

```
(defmethod =translate((p $Polygone) dx dy)
  (dolist (m (?sommets p))
    (=translate m dx dy)
  )
)
```

SYM

La base pour un point :

```
(defmethod =symx ((p $Point))
  (setf (?y p) (- 0 (?y p)))
  (describe p)
)

(defmethod =symy ((p $Point))
  (setf (?x p) (- 0 (?x p)))
  (describe p)
)

(defmethod =symO ((p $Point))
  (=symx p)
  (=symy p)
  (describe p)
)
```

Pour objet de Triangle et Polygone, juste faire tous les points SYM.

Mais, il faut faire attention à SYM de Carré et Rectangle :

```

(defmethod =symx ((c carre))
  (setf (y? c) (+ (- 0 (y? c)) (longueur? c)))
  (describe c))

(defmethod =symy ((c carre))
  (setf (x? c) (- 0 (x? c) (longueur? c)))
  (describe c))

(defmethod =symO ((c carre))
  (=symx c)
  (=symy c)
  (describe c))

```

Zoom

Le point central ne bouge pas, on modifier la position du sommet avec la fonction Translate.

```

(defmethod =zoom ((c cercle) z )
  (setf (rayon? c) (* z (rayon? c)))
  (describe c))

(defmethod =zoom ((c carre) z )
  (setf (longueur? c) (* z (longueur? c)))
  (=translate c (- 0 (* z(longueur? c) (/ (sqrt 2) 2))) (- 0 (* z (longueur? c) (/ (sqrt 2) 2))))
  (describe c))

```

Duplicate

```

(defmethod =duplicate ((p point) dx dy)
  (make-instance 'point
    :x (+ (x? p) dx)
    :y (+ (y? p) dy)))
(defmethod =duplicate ((c cerle) dx dy)
  (make-instance 'cerle
    :x (+ (x? c) dx)
    :y (+ (y? c) dy)
    :rayon (rayon? c)))
(defmethod =duplicate ((c carre) dx dy)
  (make-instance 'carre
    :x (+ (x? c) dx)
    :y (+ (y? c) dy)
    :longueur (longueur? c)))
(defmethod =duplicate ((p polygone) dx dy)
  (make-instance 'polygone :sommets (loop for i in (sommets? p) collect (=duplicate i dx dy))))

```

